CS61B Lecture 16

Friday, February 28, 2020

> "An engineer is someone who can do for a dime what any fool could do for a dollar"

Cost is a principal concern throughout engineering, where cost can mean:

- operational cost (for programs, time to run, space requirements)
- development cost
- maintenance cost
- costs of failure: how robust? how safe?

Is this program fast enough? This answer will depend on for what purpose, and for what input data our program is.

How much space will our program take? And most importantly, for this class, how well will your program scale, as the input increases?

Example

Scan a text corpus (say 108 bytes or so), and find and print the 20 most frequently used words, together with counts of how often they occur.

- Solution 1 (Knuth): Heavy-duty data structures
  Knuth devised a little system for writing beautifully documented large programs, and did a little data structure "hacking" using implementations of a Hash Trie (which we will see later), with randomized placements, pointers galore. The program is several pages long.
- Solution 2 (McIlroy): UNIX shell script
  McIlroy used the built-in features of the Unix shell to write a program that fits in five lines:

  ```
  tr -c -s '[:alpha:]' '[\n*]' < FILENAME | \
  sort | \
  uniq -c | \
  sort -n -r -k 1,1 | \
  sed 20q
  ```

The question fundamentally becomes: which program is better?

Solution 1 is much faster, and there's no question that it's more elegant.

But solution 2 took 5 minutes to write, and doesn't exactly count for slow: it processes 100 MB in 50 seconds.

It can be argued that in very many cases, solution 2 is *better*, because it keeps it simple, and from the engineering perspective of development cost, it is much lower.

Measures of Cost

Time

There are a few different ways we can measure the cost of a program in terms of time.

One way is the **wall-clock** time, or **execution** time. You can do this yourself:

```
time java FindPrimes 1000
```

This is fast, easy to measure, and the meaning is obvious. It's appropriate where time is critical (in real-time systems, for example), but it also applies only to that specific data set, compiler, machine and other conditions.

We can also measure the **dynamic statement count** of the number of times statements are executed. This is more general -- it's not sensitive to the speed of the machine -- but it doesn't tell you the actual time, and still only applies to that specific data set.

The **symbolic execution time** is the formula for execution times as a function for input size. This applies for all inputs and make scaling clear, but the practical formula must be approximated, and may actually tell very little about the actual time.

Asymptotic Cost

Symbolic execution time lets us see the shape of the cost function. Since we are approximating anyway, it is pointless to be precise about these things:

- Behavior on small inputs
  - We can always pre-calculate some results.
  - Times for small inputs is usually not important.
  - We are usually much more interested in **asymptotic behavior** as the input size becomes very large.
- Constant factors
  - Just changing machines can cause constant-factor change.
  - An algorithm that is fundamentally fast is not worse just because it is run on a slower machine.

How can we abstract away these things?

Order Notation

The idea is that we shouldn't try to produce specific functions that specify size, but rather produce **families of functions** with similarly behaved magnitudes.

This notation is not specific to computer science, but instead borrowed from mathematics.

For the purposes of this section, we say that `f` is bounded by `g` if it is in `g`'s family.

For any function

$$g(x)$$

the functions

$$2g(x), 0.5g(x), \ldots, K \bullet g(x), \forall K \in \mathbb{R}^+$$

all have the same "shape", so we put them into `g`'s family.

Any function

$$h(x)$$

such that

$$h(x) = K \bullet g(x), \forall x > M, M \in \mathbb{R}$$

has `g`'s shape except for small values, is still in `g`'s family.

To calculate the **upper limit**, we throw in all functions whose absolute value is everywhere smaller than or equal to some member of `g`'s family. We call this set:

$$O(g), or\ O(g(n))$$

In mathematical notation, this is written as:

$$f(n) = O(g(n))$$
$$\exists C, N \Rightarrow |f(n)| \leq Cg(n), \forall n \geq N$$

For **lower limits**, we throw in all functions whose absolute value is everywhere greater than or equal to some member of `g`'s family. We call this set:

$$\Omega(g)$$

In mathematical notation:

$$f(n) = \Omega(g(n))$$
$$\exists C, N \Rightarrow |f(n)| \geq Cg(n), \forall n > N$$

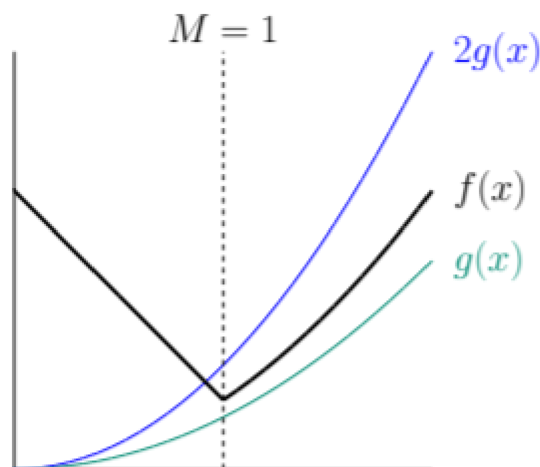
And finally, we define

$$\Theta(N) = O(g) \cap \Omega(g)$$

as the set of functions bracketed in magnitude by two members of `g`'s family.

Illustration

Big O



Here,

$$f(x) \leq 2g(x), \forall x > 1$$

We say that `f(x)` is in `g`'s bounded above family, written as:

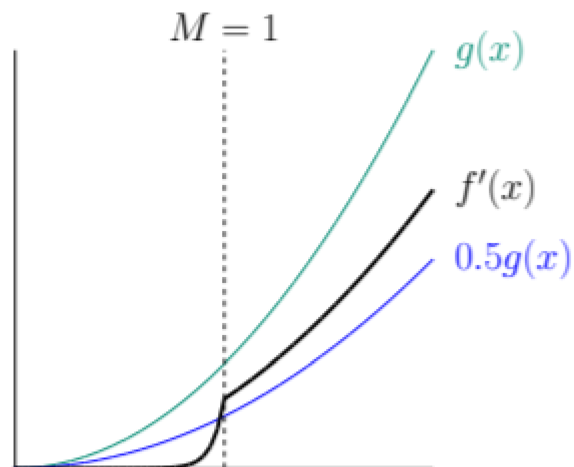
$$f(x) \in O(g(x))$$

even though, in this case,

$$f(x) > g(x)$$

everywhere.

Big Omega



Here,

$$f'(x) \geq \frac{1}{2}g(x), \forall x > 1$$

So `f'(x)` is in `g`'s bounded below family, written as:

$$f'(x) \in \Omega(g(x))$$

even though

$$f(x) < g(x)$$

everywhere.

Big Theta

In the previous two examples, we not only have:

$$f(x) \in O(g(x)) \text{ and } f'(x) \in \Omega(g(x))$$

but also:

$$f(x) \in \Omega(g(x)) \text{ and } f'(x) \in O(g(x))$$

We can summarize this by saying:

$$f(x) \in \Theta(g(x)) \text{ and } f'(x) \in \Theta(g(x))$$

> **Mathematical pedantry**
>
> Technically, this notation
>
> $$f(x) \in O(g(x))$$
>
> is not 100% correct, since we are describing sets, and we should be writing:
>
> $$f \in O(g)$$
>
> This is however, not worse than the standard notation used outside of CS61B:
>
> $$f(x) = O(g(x))$$

> which Professor Hilfinger thinks is a serious abuse of notation.

Why It Matters

Computer scientists often talk as if constant factors didn't matter at all, and the only difference is between

$$\Theta(N) \text{ vs. } \Theta(N^2)$$

In reality, it does matter for some values, but it does indeed get "swamped" for larger values:

n	$16 \lg n$	$\sqrt{n}$	n	$n \lg n$	n^2	n^3	2^n
2	16	1.4	2	2	4	8	4
4	32	2	4	8	16	64	16
8	48	2.8	8	24	64	512	256
16	64	4	16	64	256	$4,096$	$65,636$
32	80	5.7	32	160	1024	$32,768$	4.2×10^9
64	96	8	64	384	$4,096$	$262,144$	1.8×10^{19}
128	112	11	128	896	$16,384$	2.1×10^9	3.4×10^{38}
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$1,024$	160	32	$1,024$	$10,240$	1.0×10^6	1.1×10^9	1.8×10^{308}
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
2^{20}	320	1024	1.0×10^6	2.1×10^7	1.1×10^{12}	1.2×10^{18}	$6.7 \times 10^{315,652}$

How big a problem can you solve?

Observe the following table, where the left column shows the time in microseconds to solve a given problem, as a function of problem size N, and the entries show the maximum size of problem that can be solved in a second, hour, month of 31 days, and century:

Time (μsec) for problem size N	Max N Possible in 1 second	1 hour	1 month	1 century
$\lg N$	10^{300000}	$10^{1000000000}$	$10^{8 \cdot 10^{11}}$	$10^{10^{14}}$
N	10^6	$3.6 \cdot 10^9$	$2.7 \cdot 10^{12}$	$3.2 \cdot 10^{15}$
$N \lg N$	63000	$1.3 \cdot 10^8$	$7.4 \cdot 10^{10}$	$6.9 \cdot 10^{13}$
N^2	1000	60000	$1.6 \cdot 10^6$	$5.6 \cdot 10^7$
N^3	100	1500	14000	150000
2^N	20	32	41	51

Using Notation

We can use this order notation for any kind of real-valued function, and we will use them to describe cost functions. Here is a very simple searching algorithm:

```
int find(List L, Object X) {
    int c;
    for (c = 0; L != null; L = L.next, c += 1) {
        if (X.equals(L.head)) {
            return c;
        }
    }
    return -1;
}
```

We must choose an operation that is representative of how the problem scales with size. In this case, it is the number of `.equals` tests.

If `L` has a length of `N`, then the loop does at most `N` tests; we call this the worst-case time.

In fact, the total number of instructions executed is roughly proportional to `N` in the worst case, so we can also say the worst-case time is

$$O(N)$$

regardless of the measurement units.

We use the

$$N > M$$

provision in the definition of Big O to ignore the empty list.

Warnings

It's also true that the worst-case time is

$$O(N^2), \because N \in \Omega(N)$$

because Big O bounds are loose.

The worst-case time is

$$\Omega(N), \because N \in \Omega(N)$$

but that does not mean that the loop always takes time N, or even

$$K \bullet N, \exists K$$

Instead, we are just saying something about the function that maps N into the largest possible time required to process any array of length N.

To say as much as possible about our worst-case time, we should try to give a theta bound: in this case, we can:

$$\Theta(N)$$

But again, that still tells us nothing about best-case time, which happens when we find X at the beginning of the loop. Best-case time is:

$$\Theta(1)$$

Some Examples

Nested Loops

Nested loops often lead to polynomial bounds:

```
for (int i = 0; i < A.length; i++) {
    for (int j = 0; i < A.length; j++) {
        if (i != j && A[i] == A[j]) {
            return true;
        }
    }
}
return false;
```

Clearly the worst-case time is

$$O(N^2)$$

where N is the length of A.

This loop is very inefficient, and we can make it more efficient:

```
for (int i = 0; i < A.length; i ++) {
    for (int j = i+1; j < A.length; j++) {
        if (A[i] == A[j]) { return true; }
    }
}
return false;
```

Now the worst case time is proportional to:

$$N - 1 + N - 2 + \ldots + 1 = N(N-1)/2 \in O(N^2)$$

So its asymptotic time is unchanged by the constant factor, but the loop is much more efficient.

Recursion and Recurrence

Here's a silly example of recursion. In the worst case, both recursive calls happen:

```
boolean occurs(String S, String X) {
    if (S.equals(X)) { return true; }
    if (S.length() <= X.length()) { return false; }
    return
        occurs(S.substring(1), X) ||
        occurs(S.substring(0, S.length()-1), X);
}
```

We define

$$C(N)$$

to be the worst-case cost of `occurs(S,X)` for `S` of length `N`, `X` of fixed size `N_0`, measured in the number of calls to `occurs`. Then,

$$C(N) = \{1 \quad \text{if } N \leq N_0 \backslash 2C(N-1) \quad \text{if } N > N_0$$

So, we can see that C grows exponentially:

$$C(N) = 2C(N-1)+1 = 2(2C(N-2)+1)+1 = \dots$$
$$= 2(\underbrace{\dots}_{N-N_0} 2 \bullet 1 + 1) + \dots + 1$$
$$= 2^{N-N_0} + 2^{N-N_0-1} + \dots = 2^{N-N_0+1} \in \Theta(2^N)$$

Binary Search

In the previous case, we saw things that grew very quickly. There are also cases where the growth is very slow:

```
boolean isIn(StringX, String[] S, int L, int U) {
    if (L > U) { return false; }
    int M = (L+U)/2;
    int direct = X.compareTo(S[M]);
    if (direct < 0) { return isIn(X,S,L,M-1); }
    else if (direct > 0) { return isIn(X,S,M+1,U); }
    else { return true; }
}
```

Here, the worst case time,

$$C(D)$$

(as measured by the number of calls to `compareTo`) depends on size

$$D = U - L + 1$$

We eliminate `S[M]` from consideration each time and look at half the rest. Assuming that

$$D = 2^K - 1$$

for simplicity:

$$C(D) = \begin{cases} 0, & \text{if } D \leq 0 \\ 1 + C((D-1)/2) & \text{if } D > 0 \end{cases}$$
$$= \underbrace{1 + 1 + \dots + 1}_{k} + 0$$
$$= k = lg(D+1) \in \Theta(lgD)$$

In other words, this algorithm grows logarithmically: the complexity grows more slowly than the growth of the size of problem.

Merge Sort

Here's another very common case, which you may have seen before:

```
List sort(List L) {
    if (L.length() < 2) { return L; }
    // Split L into L0 and L1 of about equal size
    L0 = sort(L0); L1 = sort(L1);
    return //Merge of L0 and L1
}
```

where merging takes proportional time to the size of its result.

Assuming that the size of L is

$$N = 2^k$$

worst-case cost function

$$C(N)$$

counting just merge time:

$$\begin{aligned} C(N) &= \begin{cases} 0, & \text{if } N < 2 \\ 2C(N-2) + N, & \text{if } N \geq 2 \end{cases} \\ &= 2(2C(N/4) + N/2) + N \\ &= 4C(N/4) + N + N \\ &= 8C(N/8) + N + N + N \\ &= N \bullet 0 + \underbrace{N + N + \ldots + N}_{k=lg(N)} \\ &= N\ lg(N) \end{aligned}$$

In general, we can say it's

$$\Theta(N\ lg(N))$$

for some arbitrary N.